Engineering Domain-Specific Languages

Barrett R. Bryant

Department of Computer and Information Sciences

University of Alabama at Birmingham

What are Domain-Specific Languages?

- In contrast to general purpose programming languages, domain-specific languages (DSLs) are designed for a specific task.
- DSLs are usually more oriented toward the application domain expert instead of software engineers.
- DSLs are the primary new computer languages that will be developed and their development will be more influenced by application domain demands than software engineering concerns.

Domain-Specific Language Properties

- Usually small, more declarative than imperative, and less expressive than GPLs. (C. Consel)
- May be executable specification language. (A. van Deursen, P. Klint, J. Visser)

Domain-Specific Language Properties (continued)

- Conceptual distance is reduced between the problem space and the language used to express the problem.
- Programming becomes simpler, easier, and more reliable.
- The amount of code which must be written is reduced.
- Productivity is increased and maintenance costs are decreased.

  (Center for Agile Technology – University of Texas at Austin)

Domain-Specific Language Properties (continued)

- A DSL provides a means by which domain experts may express (through well defined and clear syntax and semantics) the ideas within the domain, with no need of previous knowledge about general programming.
- All the concepts from the domain (no more and no less) are defined minimizing the "semantic gap" that exists between the problem's domain and the program, hiding the implementation details and providing "self- documentation" of the programs.

(LIFIA, Universidad Nacional de La Plata, Argentina)

Domain-Specific Language Properties (continued)

- DSLs, in many cases, are intended for use by non-programmers (hence, "end-user languages").
- In most cases, efficiency is not a major concern, and user convenience and conciseness is paramount.

  (S. Kamin)

Domain-Specific Language Properties (continued)

- DSLs offer substantial gains in expressiveness and ease of use compared with GPLs in their domain of application. (M. Mernik, J. Heering, A. Sloane)

Disadvantages of DSLs

- The costs of designing, implementing and maintaining a DSL.
- The costs of education for DSL users.
- The limited availability of DSLs.
- The difficulty of finding the proper scope for a DSL.
- The difficulty of balancing between domain-specificity and general-purpose programming language constructs.
- The potential loss of efficiency when compared with hand-coded software.

(van Deursen, Klint, Visser)

Widely Used DSLs

- BNF (Syntax specification)
- Excel macro language (Spreadsheets)
- HTML (Hypertext web pages)
- LaTeX (Typesetting)
- Make (Software building)
- SQL (Database queries)
- VHDL (Hardware design)

Applicability of DSLs

- DSLs have been used in various domains.
- These applications have clearly illustrated the advantages of DSLs over GPLs in productivity, reliability, maintainability and flexibility.
- However, the cost for DSL design, development and maintenance has to be taken into account.
- Without an appropriate methodology and tools this costs can be higher than savings obtained by the later use of DSL.

GPL Features of DSLs

- Values (Types)
  - Primitive (very often domain specific) and composite values
  - Many DSLs are typed, allowing the detection of specification errors.
  - Many DSLs provide, in addition to standard expressions like operators (function calls) and variable access, also built-in operators which are specific to the domain of the language.

GPL Features of DSLs (continued)

- Storage
  - Many DSLs provide user-defined variables, and thus a store.
  - DSLs uses the store not only for communication between statements but as well for communication between the program and its execution model.
  - Many DSLs provide assignment, sequencing, conditional, and iterative statements.
  - DSLs usually provide domain specific commands and might provide also domain specific conditional and iterative commands

GPL Features of DSLs (continued)

- Bindings
  - Many DSLs use bindings in order to have more concise specifications.
  - Declarations are often implicit.
- Abstractions
  - Most DSLs do not provide general-purpose abstraction mechanisms.
  - It is often possible to provide a fixed set of abstractions that are sufficient for all the applications in a domain.
  - Declarative abstraction mechanism

GPL Features of DSLs (continued)

- Abstractions
  - GPLs focus on providing powerful abstraction mechanisms, whereas a DSL strives to provide the right set of predefined abstractions and does not necessarily include abstraction mechanisms.
  - A GPL cannot possibly provide the right abstractions needed for all possible applications, but since a DSL has a restricted domain, it is possible to provide some, if not all, of the right abstractions.

GPL Features of DSLs (continued)

- Encapsulation
  - DSLs are not intended for programming-in-the-large
- Type-systems
  - Usually monomorphic since DSLs use abstractions in a restricted manner
- Sequencers
  - DSLs might have jumps (goto) and domain specific escapes

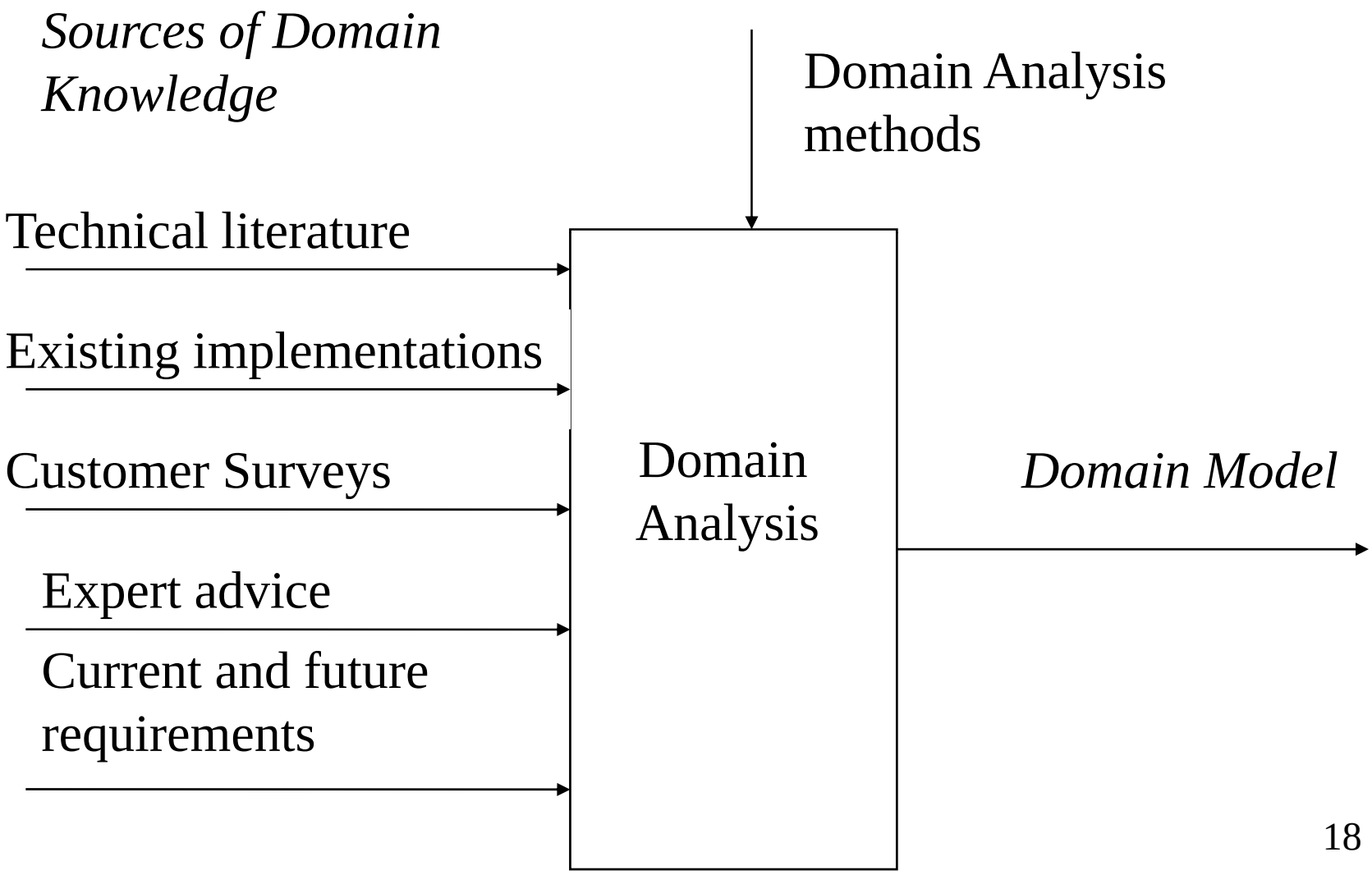
Domain Analysis

- A DSL is a programming language dedicated to a particular domain and provides appropriate built-in abstractions and notations.
  - We need to do Domain Analysis.
- Programs in DSLs explicitly specify only part of the behavior because a significant portion of the behavior is implicit and fixed.
  - We need to discover fixed and variable parts of the domain.

Domain Analysis

- *Domain*: an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.
- *Domain Analysis*: The process of identifying, analyzing and representing a **domain model** and software architecture from the study of existing systems, underlying theory, emerging technology and development histories within the domain of interest.

Domain Analysis Process



Domain Model

- A domain model is:
  - an explicit representation of the **common** and the **variable** properties of the system in a domain,
  - the semantics of the properties and domain concepts,
  - the dependencies between the properties.

Domain Model

- *Domain definition*: defines the scope of a domain and characterizes its contents by giving examples, counterexamples, and generic rules for inclusion or exclusion.
- *Terminology*: defines the domain lexicon.
- *Concept models*: describe concepts in a domain in some appropriate modeling formalism and informal text.
- *Feature models*: describe the common and the variable properties of concepts and their interdependencies. Feature models represents configuration aspects of the concept models.

Domain Analysis

- Analysis of similarity: Analyze similarities between entities, activities, events, relationships, structures, etc.
- Analysis of variations: Analyze variations between entities, activities, events, relationships, structures, etc.
- Analysis of combinations: Analyze combinations suggesting typical structural or behavioral patterns.

Feature Modeling Using FODA

- Feature models are used in Domain Analysis to capture commonalities and variabilities of systems in a domain.
- Feature models consist of:
  - Feature diagram: represents a hierarchical decomposition of features and their kinds (mandatory, alternative, optional feature)
  - Feature definitions: describe all features (semantics)
  - Composition rules for features: describe which combinations are valid/invalid
  - Rationale for features: reasons for choosing a feature
- Feature-Oriented Domain Analysis (FODA)

  http://www.sei.cmu.edu/domain-engineering/FODA.html

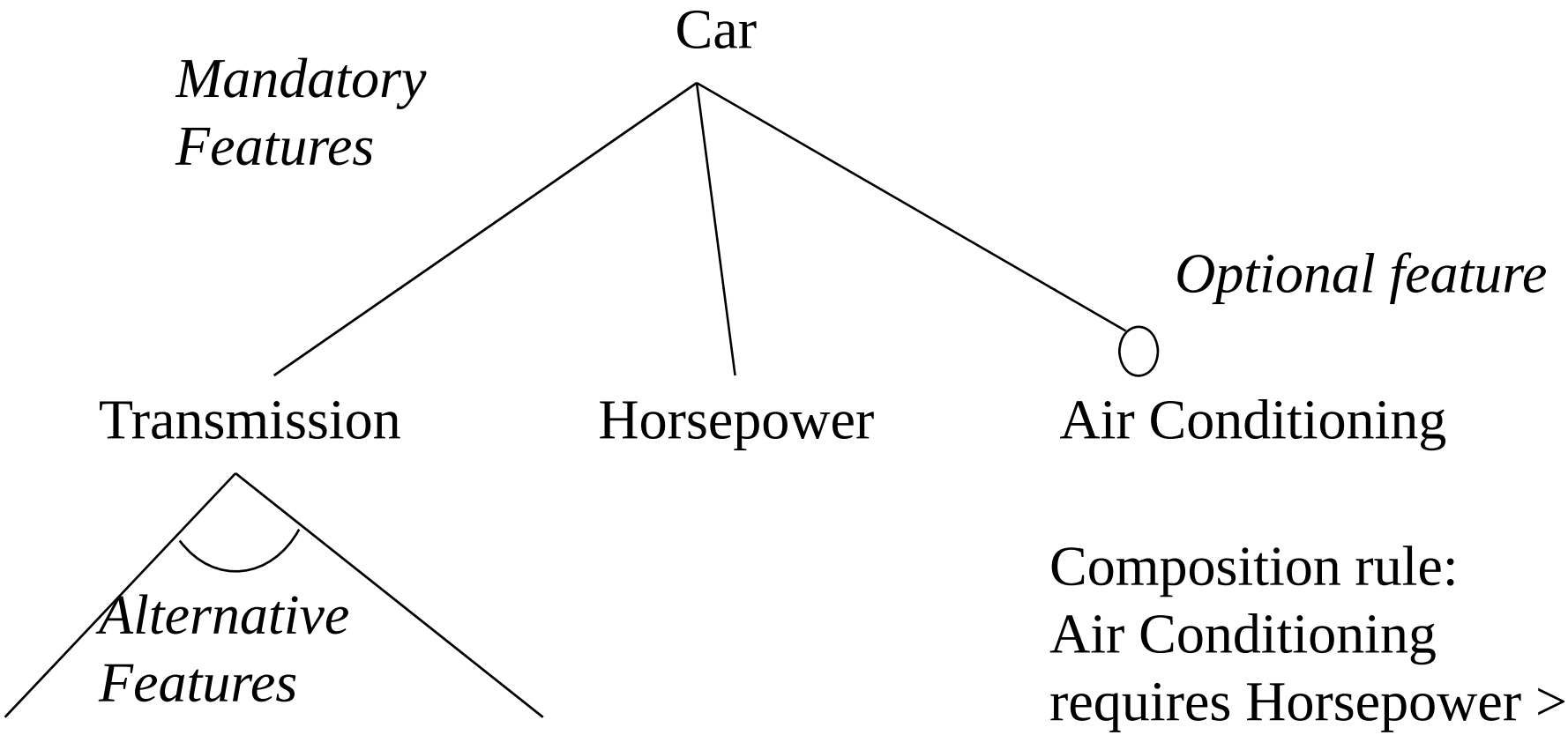
Feature Modeling Using FODA (continued)

- Mandatory features: each system in the domain must have certain features.
- Alternative features: A system can have only one feature of a time.
- Optional features: A system may or may not have certain features.
- Or-features: A system can have more than one feature.

Feature Modeling Using FODA (continued)

- Feature interdependencies are captured using composition rules:
  - Requires rules: capture implications between features
  - Mutually-exclusive rules: Model constraints on feature combinations
- Features are end-user characteristics of a system
- An instance of a feature diagram consists of an actual choice of features matching the requirements imposed by the diagram.

Example Feature Diagram



Composition rule:
Air Conditioning requires Horsepower > 100

Rationale:
Manual is more fuel efficient

FODA

- **Atomic features** cannot be further subdivided into other features.
- **Composite features** are features that are defined in terms of other features.
- **Common features** of a concept are features present in all instances of concept.
  - All mandatory features whose parent is the concept are common features.
  - Also, all mandatory features whose parents are common are themselves common features.
- Variability in the feature diagram is expressed using optional, one-of, and more-of features. These features are called also **variable features**. Nodes to which those features are attached are called **variation points**.

FODA

- Feature models are used in Domain Analysis to capture:
  - commonalities (mandatory features)
  - variabilities (optional, one-of, more-of features)
- Feature diagrams concisely describe all possible configurations (called instances) of a system, focusing on the features that may differ in each of the configuration.

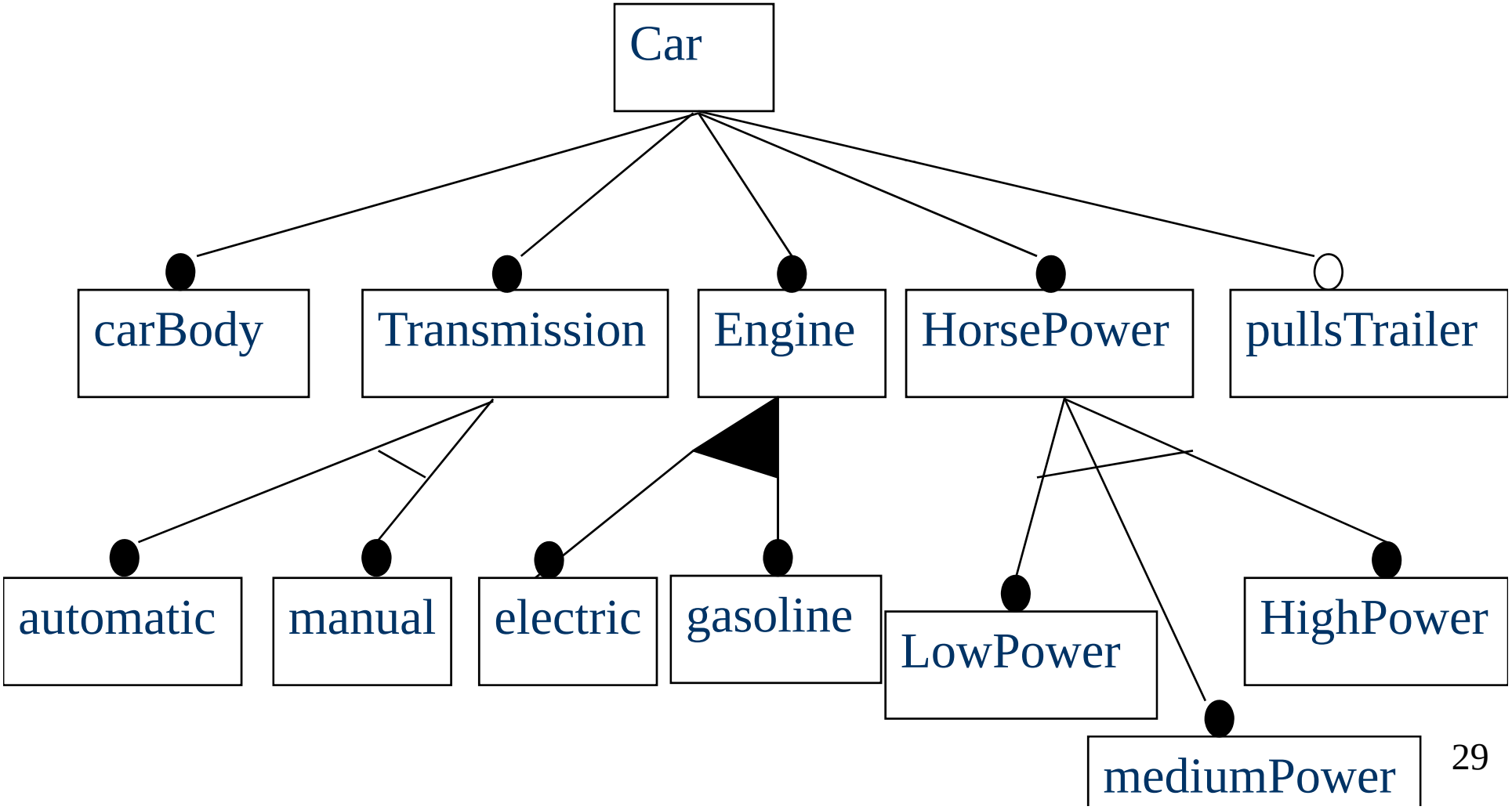
FODA

- Variability rules – count the number of possibilities for a given feature diagram

  (C–concept, A–atomic feature, F–composite)
  - Var(A)=1 // atomic feature
  - Var(F?)=var(F)+1
  - Var(one-of(F1 … Fn)) = var(F1) + …+var(Fn)
  - Var(more-of(F1 … Fn)) = (var(F1)+1) * …* (var(Fn)+1) -1
  - Var(C is F1 … Fn) = var(F1)* …* var(F2)

Example

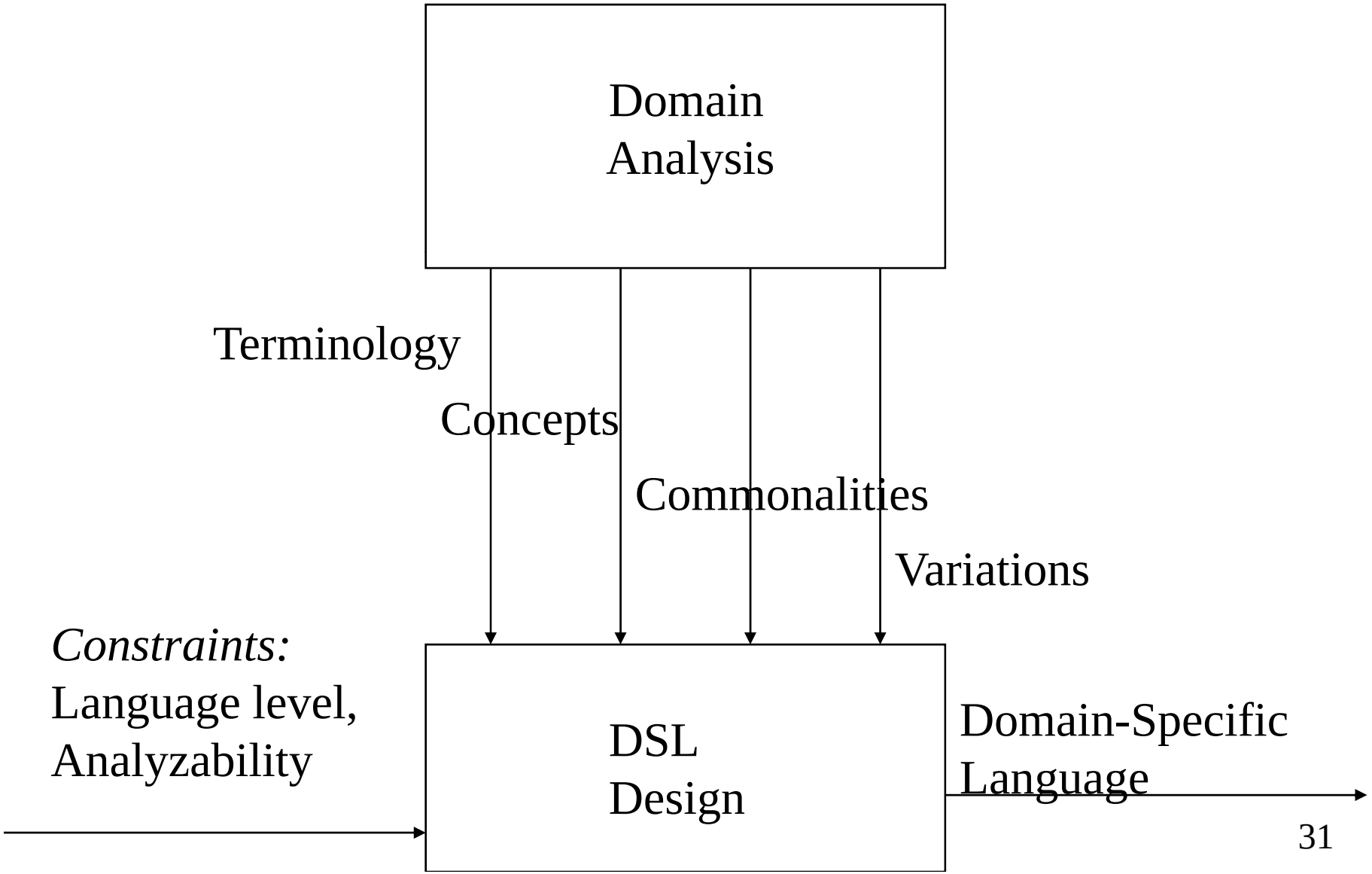
- Var(Car)=1*2*3*3*2=36



DSL Development

- Usually, during DSL development a domain analysis is done informally.
- It is highly suggested that domain analysis is done formally (e.g., FODA).
- How can a DSL be developed from the information gathered from a domain analysis?
- No clear guidelines exist!
- Some of them are presented in S. Thibault. Domain-Specific Languages. Conception, Implementation and Application, Ph.D. Thesis

DSL Design


Domain
Analysis
Terminology
Concepts
Commonalities
Variations
Constraints:
Language level,
Analyzability
DSL
Design
Domain-Specific
Language

DSL Design

- The list of variations indicate precisely what information is required to specify an instance of a system.
- This information must be **directly specified** in or **be derivable** from programs in the DSL.
- Terminology and concepts are used to guide the development of the actual DSL constructs which coresponds to the variations.
- The commonalities are used to define the execution model (through a set of common operations) and primitives of the language.

DSL Development

- Terminology
  - One desirable property of a DSL is to permit specification in a familiar notation to the domain expert.
  - Naming the abstractions in the language after the standard terminology used by domain experts is part of achieving this goal.

DSL Development

- Commonalities
  - Common parts: correspond to primitive types and operators in the language
  - Common assemblies: may correspond to the execution model or language control constructs
- Designing a language involves defining the constructs in the language and giving the semantics to the language, whether formal or informal.
- The semantics of the language describe the meaning of each construct in the language but also some fixed behavior that it is not specified by the program.
  - Execution model of a DSL is much richer than in GPL

DSL Development

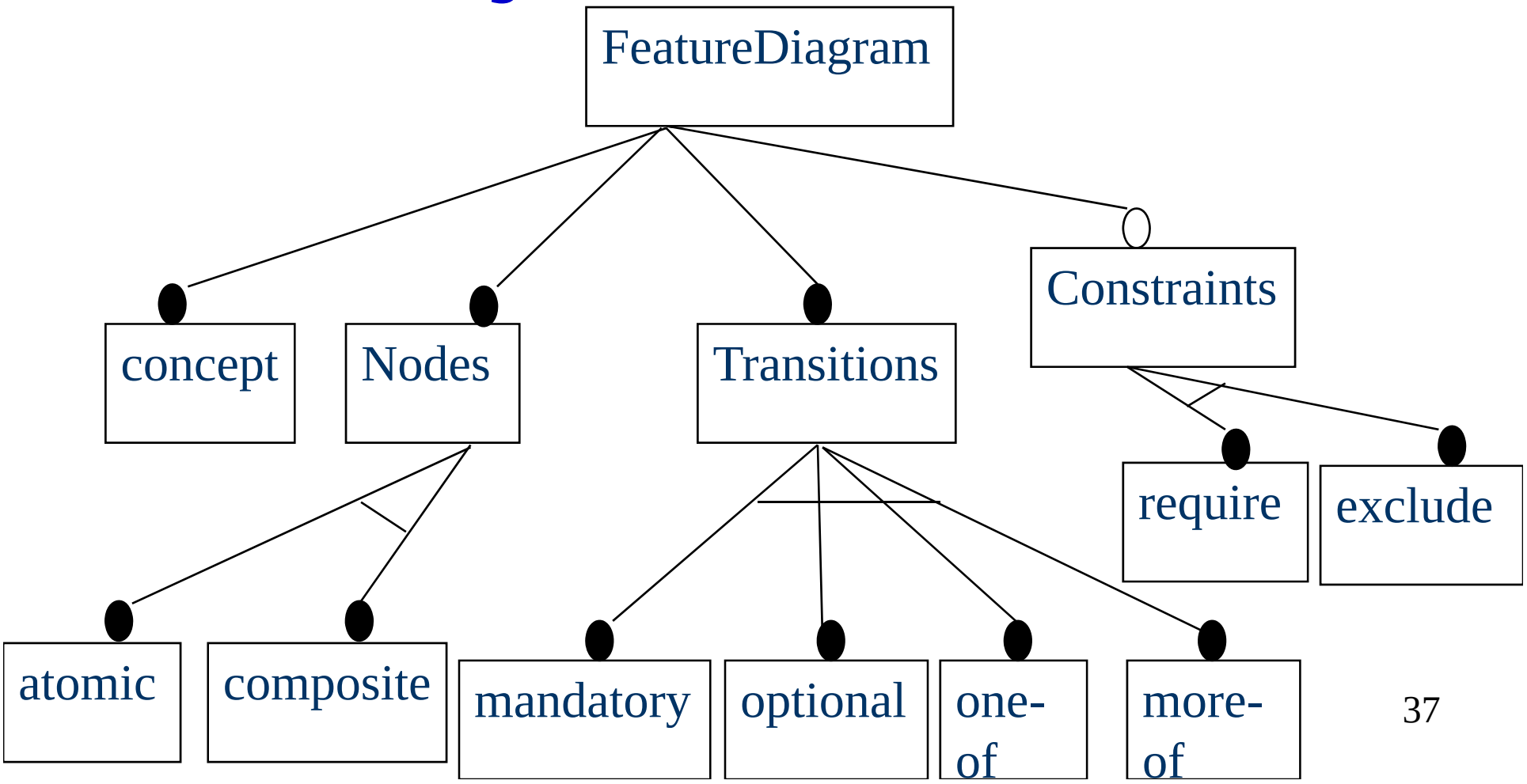
- Variations
  - Data variations – behavior that varies with respect to the values used in computations
  - Behavioral variations – behavior whose data or control flow varies
- Variations can be further classified to the range of variation:
  - Range is finite and small
  - Range is infinite or large

DSL Development

- For variations whose range is finite and small, attributes can be used to describe the variations.
  - constant definition for uniform behavior – behavior that cannot change during program execution
  - variable declaration or function/procedure definition for non-uniform behavior
  - functions/procedures may be provided whose arguments are the values to be given to the variations.
- For variations which are infinite or large, a set of operations and/or commands must be provided that can be combined to specify any behavior in the range of variations.

Example

- Let's develop a language for describing feature diagrams



Example

Car: all(carBody, Transmission, Engine, HorsePower,
pullsTrailer?)
Transmission: one-of(automatic, manual)
Engine: more-of(electric, gasoline)
HorsePower: one-of(lowPower, mediumPower, highPower)

---

car (carBody,
transmission (automatic | manual),
engine (electric + gasoline),
horsePower (lowPower | mediumPower | highPower),
trailer [pullsTrailer])

---

car (carBody, transmission, engine, horsePower, trailer)
transmission (automatic | manual)
engine (electric + gasoline)
horsePower (lowPower | mediumPower | highPower)
trailer [pullsTrailer]

DSL Design

- Approaches to DSL design can be characterized along two orthogonal dimensions:
  - the formal nature of the design description
  - the relationship between the DSL and existing languages

DSL Design

- In an informal design the specification is usually in some form of natural language probably including a set of illustrative DSL programs.
- Informal language designs can contain imprecision that causes problems in the implementation phase. They typically focus on syntax, leaving semantic concerns to the imagination.
- Formal specification of both syntax and semantics can bring problems to light before implementation.
- Formal designs can be implemented automatically by tools, thereby significantly reducing implementation effort.

DSL Design

- The easiest way to design a DSL is to base it on an existing language.
- One possible benefit is familiarity for users, but this only applies if the domain users are also programmers in the existing language, which is often not the case.
  - Existing language is partially used
  - Existing language is restricted
  - Existing language is extended

DSL Design

- If there is no relationship between the DSL and an existing language then DSL design is very similar to GPL design with additional constraints.
- The DSL designer has to keep in mind both the special character of DSLs as well as the fact that users need not be programmers.

DSL Design

- Of particular importance for DSLs are the level of abstraction the language has and the degree to which it may be analyzed.
- The abstraction level of the language is directly related to the common goals of reuse and accessibility to end-users,
- The ability to automatically analyze a DSL program and to verify or compute properties of the program is a main goal of some DSLs.

Lessons Learned from Real DSL Experiments (D. Wile, HICSS-36, 2003)

- Technological issues
- Organizational issues
- Social issues

Lessons Learned from Real DSL Experiments (D. Wile, HICSS-36, 2003)

- Lesson T1: Adopt whatever formal notations the domain experts already have, rather than invent new ones.
- Lesson T1 Corollary 1: Use their jargon terms whenever possible.
- Lesson T1 Corollary 2: One should look to informal notations of the domain as the foundation for the DSL .
- Lesson T1 Corollary 3: Adopt conventional notations, rather than invent an idiosyncratic one.

Lessons Learned from Real DSL Experiments (D. Wile, HICSS-36, 2003)

- Lesson T2: You are almost never designing a programming language. (Most DSL designers come from language design backgrounds. The admirable principles of orthogonality and economy of form are not necessarily well-applied to DSL design)
- Lesson T2 Corollary: Design only what is necessary. Learn to recognize your tendency to over-design.

Lessons Learned from Real DSL Experiments (D. Wile, HICSS-36, 2003)

- Lesson T3: Strive for an 80% solution.
- Lesson O1: Understand the organizational roles of the people who will be using your language.
- Lesson O1 Corollary 1: Understand the background expertise of the people affected by the DSL technology introduction.
- Lesson O1 Corollary 2: Understand the present solution design process thoroughly before undertaking to substitute a DSL approach.

Lessons Learned from Real DSL Experiments (D. Wile, HICSS-36, 2003)

- Lesson O2: Be sure that the intended technology transfer process from your product into their organization's infrastructure is consistent with their business model.
- Lesson S1: Find an advocate for your technology in their organization.
- Lesson S1 Corollary: Establish close ties with a domain expert to produce the infrastructure that the system will be translated into.

Lessons Learned from Real DSL Experiments (D. Wile, HICSS-36, 2003)

- Lesson S3: Do not expect the domain experts to know what the computer can (should) do for them.
- Lesson S3 Corollary 1: Do not expect the domain experts to understand what the computer cannot possibly do for them!
- Lesson S3 Corollary 2: Do not expect your users to overlook or forgive your design mistakes.

Can a DSL Implementation be Produced from Examples?

- GenParse/GenInc is a suite of grammar inference tools developed by UAB and the University of Maribor (Slovenia) to derive DSL grammars from example programs
- The tools are successful when the set of sample programs is good, especially when augmented by negative samples.
- Samples should be provided by domain experts.

Conclusions

- Domain-specific languages are promising tools to assist domain experts in specifying their programs.
- Most language development will be for DSLs, not new GPLs.
- There are promising tools to help domain experts build DSLs.

Acknowledgements

- Many slides adapted from UAB course on domain-specific languages taught by Professor Marjan Mernik.
- Additional reference:

  M. Mernik, J. Heering, A. M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys 87*, 4 (December 2005), 816-844.
- Further information:

  http://www.cis.uab.edu/softcom

  bryant@cis.uab.edu